LEARNING, DEVELOPMENT, AND PRODUCTION SYSTEMS

Robert Neches
USC/Information Sciences Institute

Pat Langley
University of California, Irvine

David Klahr
Carnegie–Mellon University

Technical Report 86–01

January 8, 1986

To appear as a chapter in D. Klahr, P. Langley & R. Neches (Eds.), *Production System Models of Learning and Development.* Cambridge, Mass.: MIT Press.

Preparation of this chapter was supported in part by a grant from the Spencer Foundation to the third author, and in part by Contract N00014–85–15–0373 from the Personnel and Training Research Program, Office of Naval Research to the second author. We would like to thank Mike Rychener and Jeff Schlimmer for comments on an earlier version of the chapter.

Introduction

The fields of cognitive science and artificial intelligence attempt to understand the *mechanisms* underlying intelligent behavior. However, many different approaches are consistent with this general goal, and various frameworks have been proposed for explaining intelligence. In this book we focus on one such approach, known as *production systems*, that has received considerable attention both within cognitive psychology and artificial intelligence. Moreover, we will limit our attention to production system models of learning and development, since this has been an active area of research that we feel has considerable long-term significance.

In the current chapter, we will introduce the production system framework and argue for the importance of studying learning phenomena. Both the chapter and the book are based on the assumption that our understanding of the psychological phenomena can be advanced by constructing and evaluating computational models of intelligent behavior. Although some readers may question this assumption, we do not have the space to defend it here. However, we refer interested parties to treatments of the issue by Newell & Simon (1972) and Anderson (1976).

Even when applied to relatively circumscribed domains, and even without the added complexity of self-modifiability, production systems are difficult to formulate and to understand. It is not surprising, therefore, that after almost two decades, only a handful of psychologists have attempted to use them. For some investigators, production systems have a forbidding aura of esoteric mystery and complexity. For others, they seem to represent an unconstrained proliferation of arbitrary assumptions and idiosyncratic notations. The "cost" of production systems is obviously high, and the "benefits" have not been immediately apparent.

In this chapter, we will attempt to clarify the benefit and, to some extent, reduce the cost by explaining the basics of production systems and discussing their relation to other areas of psychology and artificial intelligence. As editors, we are well aware that many of the subsequent chapters in this book are not easy reading. However, we will make a case for why they are worth the reader's effort. The central argument has two parts. First, learning and development are the fundamental issues in human psychology. Second, self-modifying production systems, although admittedly complex entities, represent one of the best theoretical tools currently available for understanding learning and development.

In making this argument, it is important to distinguish two related but necessarily distinct views of the role of production system models. The first framework treats production systems as a formal *notation* for expressing models. Viewed in this way, it is the content of the models, rather than the form of their expression or their interpretation scheme that is the object of interest. For example, one might characterize the rules a person uses to perform some task in terms of a production system. This type of modeling represents an attempt to formally specify the allowable "behaviors" of a system just as a grammar is intended to define the legal sentences in a language. Other formalisms for expressing the same content are possible (such as scripts, LISP programs, flow-charts, and so on), and one can debate their relative merits (c.f., Klahr & Siegler, 1978).

In contrast, the second view treats the interpreter of a production system[1] as a highly specific theory about the architecture of the human information-processing system. In its strongest form, this view asserts that humans actually employ the functional equivalent of productions in reasoning, understanding language, and other intelligent behavior. This second view also attributes great importance to the ability of production systems models to modify themselves in ways that capture many of the central features of learning and development.

We believe that it is the second view, originally put forward by Newell (1967) and most extensively applied by Anderson (1983), that provides the major justification for the use of production systems in modeling human behavior. In other words, if we are simply interested in the content of a domain, then in many cases the complexity of a production system formulation may just not be worth the effort. On the other hand, if we view the production system framework as a serious assertion about the fundamental organization of performance and learning processes, then the effort is justified.

Even the earliest production system models were designed to capture the characteristic features of the human cognitive architecture. Efforts to explore those characteristics at deeper and more concrete levels led – not surprisingly – to the discovery of technical and theoretical problems concerning management of search and control of attention. Efforts to address those problems, both within the community of production system builders and the closely related community of rule-based expert system builders, have fed back into the search for architectures that more closely reflect properties of human cognition. Issues of learning and human development have been a major forcing function in the definition of the problems, and have served as a crucial testing ground for proposed solutions. The resulting architectures have implications not only for learning, but also for the broader nature of intelligent behavior and the structure of intelligent systems.

We begin the chapter with an introduction to production systems, using a simple example from the domain of multi-column subtraction, and follow this with a brief history of the development of production systems in psychology. We then consider some phenomena of learning and development, and review some of the mechanisms that have been proposed to account for these phenomena within the production system framework. In closing, we discuss some implications of learning issues for production system models.

[1] We will use the term "production system" is used in two distinct senses. The first refers to a *theory* of the cognitive architecture, independent of particular programs that are implemented within this framework. The second refers to specific sets of condition-action rules that run within such an architecture. In general, the intended meaning should be clear from the context. Within artificial intelligence, the term "production system" sometimes includes backward chaining rule-based systems, such as that used in MYCIN (Shortliffe, 1976). However, we will use the term in the more limited (forward chaining) sense.

An Overview of Production Systems

Before moving on to the use of production systems in modeling learning and development, we should first introduce the reader to the concepts and mechanisms on which they are based. In this section we outline the basic components of production system architectures, and follow this with an example of a simple production system program for solving multi-column subtraction problems. After these preliminaries, we review some of the arguments that have been made in favor of production system models and consider exactly what we mean by a production system "architecture".

The Components of a Production System

The basic structure of production system programs (and their associated interpreter) is quite simple. In its most fundamental form, a production system consists of two interacting data structures, connected through a simple processing cycle:

- A *working memory* consisting of a collection of symbolic data items called working memory *elements*;
- A *production memory* consisting of condition-action rules called *productions*, the conditions of which describe configurations of elements that might appear in working memory, and the actions of which specify modifications to the contents of working memory;
- Production memory and working memory are related through the *recognize-act* cycle. This consists of three distinct stages:
    - The *match* process, which finds productions whose conditions match against the current state of working memory; the same rule may match against memory in different ways, and each such mapping is called an *instantiation.*
    - The *conflict resolution* process, which selects one or more of the instantiated productions for application;
    - The *act* process, which applies the instantiated actions of the selected rules, thus modifying the contents of working memory.

The basic recognize-act process operates in cycles, with one or more rules being selected and applied, the new contents of memory leading another set of rules to be applied, and so forth. This cycling continues until no rules are matched or until an explicit halt command is encountered. Obviously, this account ignores many of the details, as well as the many variations that are possible within the basic framework, but it conveys the basic idea of a production system.

A Production System for Subtraction

Now that we have considered production systems in the abstract, let us examine a specific example from the domain of arithmetic. Multi-column subtraction problems are most naturally represented in terms of rows and columns, with each row-column pair containing a number or a blank. The basic operators in subtraction involve finding the difference between two numbers in a given column, decrementing the top number in a column, and adding ten to the top number in a column. However, the need to borrow forces some sort of control symbols or goals, in order to distinguish between the column for which one is currently trying to find an answer and the column from which one is currently trying to borrow.

Before considering our production system for subtraction, we should consider the representation used for elements in working memory against which our rules will match. Suppose we have the problem 87 – 31, before the system has started trying to find an answer. We will represent this as a number of separate propositions, one for each number. For instance, the fact that the number 8 occurs in the top row and the leftmost column would be stored as (8 in column-2 row-1), in which the names for columns and rows are arbitrary. The positions of the remaining numbers would be represented by the elements (3 in column-2 row-2), (7 in column-1 row-1), and (1 in column-1 row-2). Since the position for each column's result is blank, the elements (blank result-for column-2) and (blank result-for column-1) would also be present.

Three types of relations must also be stored in memory. The first two involve the spatial relations *above* and *left-of*, and are used in elements like (row-1 above row-2) and (column-2 left-of column-1). These translate into English as "row-1 is above row-2" and "column-2 is left of column-1", respectively. Naturally, the arguments of *above* are always rows, while the arguments of *left-of* are always columns. The final predicate is the *greater-than* relation, which takes two numbers as its arguments. This relation occurs in elements such as (3 greater-than 1), which means "3 is greater than 1".[2]

One must also be able to distinguish between the column for which we are currently computing a result and the column which is the current focus of attention. We will use the label *processing* for the first and the label *focused-on* for the second. Since one always starts subtraction problems by working on the rightmost column, and since the initial focus of attention also resides there, we would begin with the additional elements (processing column-1) and (focused-on column-1).

Now let us examine the productions which operate upon this representation. We will present an English paraphrase of each rule and consider the role played by each condition and action. Italicized terms in the paraphrased rules stand for variables, which can match against any symbol as long as they do so consistently across conditions. In our discussion

[2] Since there are 10 digits, some 45 binary greater-than relations exist. Rather than storing these explicitly in working memory, one might insert a LISP function in the condition side to test whether this relation is met. Although this is not psychologically plausible, it considerably simplifies the implementation.

of a given production, we will refer to specific conditions by their position in the rule. E.g., the parenthetical expression (2) will stand for the second condition in a production. After we have described each of the rules individually, we will examine the manner in which they interact during two sample runs.

The most basic action in subtraction problems involves finding the difference between two digits in the same column. The FIND-DIFFERENCE rule is responsible for implementing this behavior, and matches when the column currently being processed (1) contains a top number (2, 4) which is greater than or equal to (5) the bottom number (3, 4). Its actions include computing the difference between the two numbers, and writing the difference as the result for that column. We are assuming that the system has the primitive capability of correctly finding the difference between two digits.

FIND-DIFFERENCE
If you are processing *column*,
  and *number1* is in *column* and *row1*,
  and *number2* is in *column* and *row2*,
  and *row1* is above *row2*,
  and *number1* is greater than or equal to *number2*,
then compute the difference of *number1* and *number2*,
  and write the result in *column*.

Once the result for the current column has been filled in with a digit (i.e., when the current column has been successfully processed), it is time to move on. If there is a column to the left of the current column (3), then the rule SHIFT-COLUMN matches, shifting processing and the focus of attention to the new column. Thus, this rule is responsible for ensuring that right to left processing of the columns occurs.

SHIFT-COLUMN
If you are processing *column1*,
  and you are currently focused on *column1*,
  and the result in *column1* is not blank,
  and *column2* is left of *column1*,
then note that you are now processing *column2*,
  and note that you are focusing on *column2*.

A special operator is needed for cases in which the bottom row in a column is occupied by a blank (since one can only find differences between numbers). The FIND-TOP rule handles this situation, matching in any case involving a number above a blank (2, 3, 4) in the column being processed (1). Upon application, the rule simply stores the top number as the result for that column. This will only occur in the leftmost columns of certain problems, such as 3456 – 21.

FIND-TOP
If you are processing *column*,
  and *number* is in *column* and *row1*,
  and *column* and *row2* is blank,
  and *row1* is above *row2*,
then write *number* as the result of *column*.

In cases where the top number in a column is smaller than the bottom number, the ADD-TEN rule must add ten to the top number (2, 3) before the system can find an acceptable difference for the column currently in focus (1). However, this rule will only match if the system has just applied its DECREMENT operator (4), and finding a column from which one can decrement is not always a simple matter, as we will see.

ADD-TEN
If you are focusing on *column1*,
  and *number1* is in *column1* and *row1*,
  and *row1* is above *row2*,
  and you have just decremented some number,
then add ten to *number1* in *column1* and *row1*.

Before the ADD-TEN rule can apply, the system must first apply the DECREMENT rule. This matches in cases where the system has shifted its focus of attention away from the column currently being processed (1, 2), and the top number in the new column is greater than zero (3, 4, 5). Upon firing, the rule decrements the top number by one and returns the focus of attention to the column immediately to the right (6). A negated condition (7) prevents DECREMENT from applying more than once in a given situation.

DECREMENT
If you are focusing on *column1*,
  but you are not processing *column1*,
  and *number1* is in *column1* and *row1*,
  and *row1* is above *row2*,
  and *number1* is greater than zero,
  and *column1* is left of *column2*,
  and you have not just decremented a number,
then decrement *number1* by one in *column1* and *row1*,
  and note that you have just decremented a number,
  and note that you are focusing on *column2*.

However, before the system can decrement the top number in a column, it must first be focused on that column. The rule SHIFT-LEFT-TO-BORROW takes the first step in this direction, matching in precisely those cases in which borrowing is required. In other words, the rule applies if the system is focused on the column it is currently processing (1, 2), if that column contains two numbers (3, 5), if the bottom one larger than the top one (4, 6), and if there is another column to the left (7) from which to borrow. A negated condition (8) keeps this rule from matching if DECREMENT has just applied, thus avoiding and

infinite loop. Under these conditions, SHIFT-LEFT-TO-BORROW shifts the focus of attention to the adjacent column.

SHIFT-LEFT-TO-BORROW
If you are focused on *column1*,
  and you are processing *column1*,
  and *number1* is in *column1* and *row1*,
  and *number2* is in *column1* and *row2*,
  and *row1* is above *row2*,
  and *number2* is greater than *number1*,
  and *column2* is left of *column1*,
  and you have not just decremented a number,
then note that you are focusing on *column2*.

For many borrowing problems (such as 654 – 278), the SHIFT-LEFT-TO-BORROW rule is quite sufficient, since one can borrow from the column immediately to the left of the one being processed. However, if the column to the left contains a zero as its top number, one must search further. The rule SHIFT-LEFT-ACROSS-ZERO handles cases such as these, matching if the system has shifted its focus of attention (1, 2), if it has found a zero (3) in the top row (4) of the new column, and if there is another column to the left (5). When these conditions are met, the rule shifts attention to the column immediately to the left. Again, this will not occur if DECREMENT was just applied (6), since this would cause an infinite loop. On problems involving multiple zeros in the top row (such as 10005 – 6), SHIFT-LEFT-ACROSS-ZERO will apply as many times as necessary to reach a number which it can decrement.

SHIFT-LEFT-ACROSS-ZERO
If you are focused on *column1*,
  and you are not processing *column1*,
  and the number in *column1* and *row1* is zero,
  and *row1* is above *row2*,
  and *column2* is left of *column1*,
  and you did not just decrement a number,
then note that you are focusing on *column2*.

The above rules handle the major work involved in multi-column subtraction, but the system must also have some way to know when it has completed solving a given problem. The rule FINISHED is responsible for noting when there is no longer a blank (3) in the result position of the left-most column (1, 2). In such cases, this rule fires and tells the production system to halt, since it has generated a complete answer to the problem.

FINISHED
If you are processing *column1*,
  and there is no *column2* to the left of *column1*,
  and the result of *column1* is not blank,
then you are finished.

A Simple Subtraction Problem

Now that we have examined each of the rules in our system, let us consider its behavior on some specific subtraction problems. Presented with the problem 87 – 31, the system begins with the problem representation we described earlier.[3] This consists of 10 basic working memory elements:

(8 in column-2 row-1)
(7 in column-1 row-1)
(3 in column-2 row-2)
(1 in column-1 row-2)
(processing column-1)
(focused-on column-1)
(blank result-for column-2)
(blank result-for column-1)
(column-2 left-of column-1)
(row-1 above row-2)

We are assuming here that greater-than relations are tested by LISP predicates rather than being explicitly represented in memory, while the relations above and left-of are stored directly. Note that the rightmost column (column-1) is labeled for processing first, and that this column is also the initial focus of attention.

Given this information, the rule FIND-DIFFERENCE matches against the elements (processing column-1), (7 in column-1 row-1), (1 in column-1 row-2), (row-1 above row-2), and (7 greater-than 1). The action side of this production is instantiated, computing the difference between 7 and 1, and the number 6 is "written" in the results column. This last action is represented by the new element (6 result-for column-1).

Now that there is a result in the current column, the rule SHIFT-COLUMN matches against the elements (processing column-1), (focused-on column-1), and (column-2 left-of column-1). Although this production has four conditions, only the first, second, and third are matched, since the third is a negated condition that must not be met. Upon application, the rule removes the elements (processing column-1) and (focused-on column-1) from memory, and adds the elements (processing column-2) and (focused-on column-2). In other words, having finished with the first column, the system now moves on to the

[3] This representation may be provided directly by the programmer, or it may be generated automatically by the system from linear input, using another set of rules. In any case, we will ignore this preprocessing stage to keep our example as simple as possible.

adjacent one.

At this point, the conditions of FIND-DIFFERENCE are again met, though this time they match against a different set of elements – (processing column-2), (8 in column-2 row-1), (3 in column-2 row-2), (row-1 above row-2), and (8 greater-than 3). As before, the rule computes the difference of the two digits, and adds the element (5 result-for column-2) to working memory. The presence of this new element serves to trigger the conditions of FINISHED, since both columns now have answers. Having generated a complete answer to the problem (56), the system halts.

A Complex Subtraction Problem

Since the above problem involved no borrowing, we saw only a few of the rules in action, so let us now turn to the more difficult problem 305 – 29. At the outset of this task, working memory contains the following elements:

(blank in column-5 row-2)
(3 in column-5 row-1)
(column-5 left-of column-4)
(2 in column-4 row-2)
(0 in column-4 row-1)
(column-4 left-of column-3)
(9 in column-3 row-2)
(5 in column-3 row-1)
(focused-on column-3)
(processing column-3)
(row-1 above row-2)

If we measure complexity by the number of elements in memory, this problem is only slightly more complex than our first example. However, the manner in which these elements interact with the various productions leads to significantly more processing than in the earlier case.

As before, the system begins by attempting to process the rightmost column. Since the greater-than condition of FIND-DIFFERENCE is not met (5 is not greater than 9), this rule can not apply immediately. Instead, the production SHIFT-LEFT-TO-BORROW matches, and shifts the focus of attention to the column to the left (column-3). On many borrowing problems, the system would apply DECREMENT at this point, but the conditions of this rule are not met either (zero is not greater than zero). However, the conditions of SHIFT-LEFT-ACROSS-ZERO are met, and this leads to system to move its focus even further left, to column-5.

Finally the system has reached a column in which it can apply DECREMENT, and in doing so it replaces the element (3 in column-5 row-1) with the element (2 in column-5 row-1). This rule also moves the focus back one column to the right, making column-4 the center of attention. Given this new situation, ADD-TEN matches and the 0 in the middle column is replaced with the "digit" 10. This is accomplished by replacing the element (0

in column-4 row-1) with the new element (10 in column-4 row-1).

Now that the system has a number larger than zero in the middle column, it can decrement this number as well. The DECREMENT production matches against the current state of memory, replacing the recently generated 10 with the digit 9. In addition, the rule shifts the focus of attention from the middle column (column-4) back to the rightmost column (column-3). Since the system has just decremented a number in the adjacent column, the rule ADD-TEN now replaces the 5 in the rightmost column with the "digit" 15. This in turn allows FIND-DIFFERENCE to apply, since the number in the top row of this column is now larger than the number in the bottom row. The effect is that the difference 6 (that is, 15 – 9) is added as the result for column-3.

The presence of a result in the current column causes SHIFT-COLUMN to match, and both the processing marker and the focus of attention are shifted to the center column. Since the top number in this column is larger than the bottom number (from our earlier borrowing), FIND-DIFFERENCE matches and computes 9 – 2. The resulting difference 7 is "written" as the result for the middle column, and this in turn leads SHIFT-COLUMN to fire again, shifting processing and attention to the rightmost column.

In this case the bottom row of the current column is blank, so the only matched production is FIND-TOP. This rule decides that the result for the rightmost column should be 2 (the digit in the top row), and since all columns now have associated results, the rule FINISHED applies and the system halts, having computed the correct answer of 276.

Comments on Subtraction

Although we hope that our program for subtraction has given the reader a better feel for the nature of production system models, it is important to note that it makes one major simplification. Since only one rule (and only one instantiation of each rule) matches at any given time, we were able to ignore the issue of conflict resolution. Although this was useful for instructional purposes, the reader should know that there are very few tasks for which it is possible to model behavior in such a carefully crafted manner, and conflict resolution has an important role to play in these cases.

Our choice of subtraction as an example gives us some initial insight into the advantages of this framework, which we consider in more detail below. Brown & Burton (1978) and Brown & VanLehn (1980) have made significant strides in classifying children's subtraction errors, and in accounting for these errors using their "repair theory". However, Young & O'Shea (1981) have proposed an alternative model of subtraction errors based on a production system analysis. They account for many of the observed errors by omitting certain rules from a model of correct behavior. Langley & Ohlsson (1984) have taken this approach further, accounting for errors only in terms of incorrect *conditions* on the rules shown above. Within this framework, they have developed a system that automatically constructs a model to account for observed errors. This is possible only because of the inherent modularity of production system programs.

Before closing, we should point out one other feature of the production system approach – it does not completely constrain the systems one constructs, any more than other

computer programming languages. For example, Anderson (1983) has presented a production system model of multi-column addition that is organized quite differently from our subtraction system, despite the similarity of the two tasks. The point is that there is considerable room for different "programming styles" within the production system framework, since people can employ quite different representations and control structures. Thus, Anderson's model makes heavy use of explicit goals that are held in working memory; our "processing" and "focus" elements play a similar role, but organize behavior in a different fashion.

Advantages of Production System Models

Now that we have seen an example of a production system model, it is appropriate to consider the advantages of this approach. The fact that production systems allow one to carry out subtraction or any other behavior is not sufficient – most programming languages are equivalent in computational power. However, Newell & Simon (1972) have argued that production systems have a number of features that recommend them for modeling human behavior. Let us recount some of these characteristics:

- *Homogeneity.* Production systems represent knowledge in a very homogeneous format, with each rule having the same basic structure and carrying approximately the same amount of information. This makes them much easier to handle than traditional flow diagram models.
- *Independence.* Production rules are relatively independent of each other, making it easy to insert new rules or remove old ones. This makes them very useful for modeling successive stages in a developmental sequence, and also makes them attractive for modeling the incremental nature of much human learning.
- *Parallel/serial nature.* Production systems combine the notion of a parallel recognition process with a serial application process; both features seem to be characteristic of human cognition.
- *Stimulus-response flavor.* Production systems inherit many of the benefits of stimulus-response theory, but few of the limitations, since the notions of stimuli and responses have been extended to include internal symbol structures.
- *Goal-driven behavior.* Production systems can also be used to model the goal-driven character of much human behavior. However, such behavior need not be rigidly enforced; new information from the environment can interrupt processing of the current goal.
- *Modeling memory.* The production system framework offers a viable model of long-term memory and its relation to short-term memory, since the matching and conflict resolution process embody principles of retrieval and focus of attention.

Taken together, the above features suggest the production system approach as a useful framework within which to construct models of human behavior. Of course, this does not mean that production systems are the *only* such framework, but there seems sufficient promise to pursue the approach vigorously.

As we will shortly see in the historical section, production systems have been used successfully to model human behavior on a wide range of tasks. As we will also see, Newell & Simon's initial ideas on the nature of the human cognitive architecture have been revised along many dimensions, and many additional revisions will undoubtedly be required before we achieve significant understanding of human learning and performance. This observation leads us back to the issue of viewing production systems as cognitive architectures, to which we now turn.

Production Systems as Cognitive Architectures

The term "cognitive architecture" denotes the invariant features of the human information processing system. Since one of the major goals of any science is to uncover invariants, the search for the human cognitive architecture should be a central concern of cognitive psychology. The decision to pursue production system models involves making significant assumptions about the nature of this architecture, but within the boundaries defined by these assumptions there remains a large space of possibilities.

In later chapters, we will see many different instantiations of the basic production system framework, but let us anticipate some of these variations by considering the dimensions along which production system architectures may differ:

- *Working memory issues:*
  - *The structure of memory.* Is there a single general working memory, or multiple specialized memories (such as data and goal memories)? In the latter case, which are matched against by production memory?
  - *The structure of elements.* What is the basic form of working memory elements (e.g., list structures, attribute-value pairs)? Do elements have associated numeric parameters, such as activation or recency?
  - *Decay and forgetting.* Are there limits on the number of items present in working memory? If so, are these time-based or space-based limitations?
  - *Retrieval processes.* Once they have been "forgotten", can elements be retrieved at some later date? If so, what processes lead to such retrieval? E.g., must productions add them to memory, or does "spreading activation" occur?
- *Production memory issues:*
  - *The structure of memory.* Is there a single general production memory, or are there many specialized memories? In the latter case, are all memories at the same level, or are their organized hierarchically?
  - *The structure of productions.* Do productions have associated numeric parameters (e.g., strength and recency) or other information beyond conditions and actions?
  - *Expressive power of conditions.* What types of conditions can be used to determine whether a rule is applicable? E.g., can arbitrary predicates be included? Can sets or sequences be matched against? Can many-to-one mappings occur?

- *Expressive power of actions.* What kind of processing can be performed within the action side of an individual rule? E.g., can arbitrary functions be evoked? Can conditional expressions occur?
- *Nature of the match process.* Are exact matches required or is partial matching allowed? Does the matcher find all matched rules, or only some of them? Does the matcher find all instantiations of a given production?

- *Conflict resolution issues:*
  - *Ordering strategies.* How does the architecture order instantiations of productions? E.g., does it use the recency of matched elements or the specificity of the matched rules?
  - *Selection strategies.* How does the architecture select instantiations based on this ordering? E.g., does it select the best instantiation, or does it select all those above a certain threshold?
  - *Refraction strategies.* Does the architecture remove some instantiations permanently? E.g., it may remove all instantiations that applied on the last cycle, or all instantiations currently in the conflict set.

- *Self-modification issues:*
  - *Learning mechanisms.* What are the basic learning mechanisms that lead to new productions? Examples are generalization, discrimination, composition, proceduralization, and strengthening.
  - *Conditions for learning.* What are the conditions under which these learning mechanisms are evoked? E.g., whenever an error is noted, or whenever a rule is applied?
  - *Interactions between mechanisms.* Do the learning mechanisms complement each other, or do they compete for control of behavior? E.g., generalization and discrimination move in opposite directions through the space of conditions.

To summarize, the basic production system framework has many possible incarnations, each with different implications about the nature of human cognition. The relevance of these issues to enduring questions in the psychology of learning and development should be obvious. The chapters in this volume represent only a small sample from the space of possible architectures, although they include many of the self-modifying production systems that have been proposed to date. Before elaborating some of the issues involved with the formulation of learning systems, we will digress slightly in order to establish some historical context for production systems.

The History of Production System Models

Although they have their roots in the formalisms of computer science and mathematics, the relevance of production systems to psychology began some 20 years ago, when Newell (1967) first proposed them as one way to formulate information processing theories of human problem solving behavior. Their initial use was to provide theoretical accounts of human *performance* on a variety of tasks, ranging from adults' behavior on various puzzles (Newell & Simon, 1972) to children's responses to class-inclusion questions (Klahr & Wallace, 1972). However, it soon became apparent that they were eminently well suited for dealing with issues of *development* (Klahr & Wallace, 1973; 1976) and *learning* (Anderson, Kline & Beasley, 1978; Anzai & Simon, 1979; Langley, Neches, Neves & Anzai, 1980). Below we review the history of these efforts, including specific models and general production system architectures.

Specific Production Systems Models

The early production system models were not implemented on a computer and required "hand simulation" in order to determine their consequences. The first running production system appears to have been Waterman's (1970) poker playing program. It is interesting to note that this was also a learning system, though it was not an attempt to model the human learning process.

Soon thereafter, Newell (1972) described the implementation of a general purpose production system interpreter (called PSG) that could be applied to a wide range of cognitive domains. Newell's first two applications of PSG were atypical of the subsequent use of production systems; the first (Newell, 1972) focused on stimulus encoding, while the second (Newell, 1973) modeled performance on the Sternberg memory scanning paradigm. Both papers presented a much finer grained analysis than the majority of production system models have employed. Klahr's (1973) model of basic quantification processes was also designed to account for very fine-grained encoding processes.

A few years later, Waterman (1975) reported new results with adaptive production systems. One of these systems implemented EPAM-like discrimination networks as production rules, and another focused on sequence extrapolation tasks. Again, neither was intended as a cognitive simulation, but both were interesting in their adaptive characteristics. One year later, Rychener (1976) completed a thesis in which he reimplemented a variety of well-known AI systems as production systems. This work convincingly demonstrated that the production system framework was as powerful as other existing representational schemes, and that it was a useful framework for implementing complex models of intelligence.

Anderson, Kline & Lewis (1977) went on to use production systems to model the complexities of natural language understanding, and Anderson (1976) also applied the approach to a variety of other information processing tasks. Thibadeau, Just & Carpenter (1982) went even further, showing how production system models could account for reaction time phenomena in reading tasks. Finally, Ohlsson (1980b) employed the production system framework to model detailed verbal protocols on transitive reasoning tasks. Since 1980, the use of production systems in modeling human performance has spread, but since

performance is not our central concern, let us turn our attention to other matters.

In addition to Waterman's early concern with learning, a number of early researchers employed production systems in modeling cognitive development. However, rather than develop models of the transition process at the outset, they instead constructed models of behavior at successive developmental stages. Klahr & Wallace (1973, 1976) carried out the earliest work of this type, constructing stage models of behavior on a variety of Piagetian tasks, including class inclusion and conservation of quantity. The basic approach involved modeling behavior at different stages in terms of production system programs that differed by only one or a few rules.

Baylor, Gascon, Lemoyne & Pother (1973) extended this approach to Piagetian length and weight seriation tasks, and Young (1976) carried out an even more detailed analysis of length seriation in his thesis research. Klahr & Siegler (1978) developed similar production system stage models for Piaget's balance scale task, combining this with detailed empirical studies of children's behavior on the task. Larkin (1981) used a comparable framework to model adult expert-novice differences in physics problem solving, and Ohlsson (1980a) modeled similar differences in ability on his transitive reasoning tasks.

These stage models were tantalizing, in that they explained behavioral differences at successive stages in terms of slightly different rule sets, and yet provided no *mechanisms* to account for the transition process. (The work by Klahr & Wallace did eventually lead to a transition model, as we will see in Chapter 8.) After Waterman's early forays, the first truly adaptive production system models were reported by Anderson, Kline & Beasley (1978); these employed mechanisms of generalization, discrimination, and strengthening, about which we will hear more later in the chapter by Langley.

Shortly thereafter, production system models of learning became an active research area. For instance, Anzai (1978) reported a model of human learning on the Tower of Hanoi puzzle, and Neves (1978) proposed a model of algebra learning. At about the same time, Langley (1978) developed an adaptive production system for concept attainment, sequence extrapolation, and simple empirical discovery, while McDermott (1979) described a production system approach to reasoning by analogy. (However, neither Langley nor McDermott's systems were concerned with modeling human behavior in any detail.) Finally, Lewis (1978) introduced the mechanism of composition, which he describes in more detail in Chapter 7.

Research on adaptive production systems continued into the 1980's, with Neves & Anderson (1981) extending Lewis' notion of composition and introducing the mechanism of proceduralization. Neches (1981a, 1981b) took a quite different approach in his theory of strategy transformation (see Chapter 4), and Newell & Rosenbloom (1981) proposed a theory of learning by chunking (see Chapter 5). Since then, research in production system models of learning has continued unabated, and many of the researchers in the new field of machine learning have taken a closely related approach.[4] In the following chapters, we will examine some of this work in more detail.

[4] For instance, nearly all work on heuristics learning has represented procedures in terms of condition-action rules or productions.

Research on Production System Architectures

Our treatment of production systems would not be complete without some discussion of the history and development of production system architectures and their implementations as programming languages. Upon examining the history of research on production systems, one is struck by an obvious trend – nearly every researcher who has developed production system models of significant complexity has developed his own architecture and associated language. However, many of these architectures are very similar, and it is worthwhile to trace their evolution over time. Figure 1 summarizes this evolutionary process.

The first widely-used production system programming language was PSG (Newell & McDermott, 1975). Rather than embodying a single architecture, PSG provided a number of parameters that defined an entire *class* of architectures. Thus the modeler could explore different memory limitations, alternative conflict resolution schemes, and similar issues. However, PSG did assume a "queue" model of working memory, with elements stored in an ordered list. Many of the early production system models were implemented in PSG, including Newell's (1973) model of the Sternberg phenomenon and Klahr & Wallace's (1976) developmental stage models.

Another early production system language was Waterman's PAS. This was not intended as a serious model of the human cognitive architecture,[5] but it did have some primitive learning capabilities. Conflict resolution was based on the order of productions, and working memory was stored as a queue. Ohlsson (1979) developed a similar production system language called PSS, which he used in his models of transitive reasoning. This was very similar to PAS in that it used production order for conflict resolution and a queue-based model of short-term memory; however, one could also divide production rules into a number of sets that were organized hierarchically.

Rychener's PSNLST (1976) incorporated a number of novel features. First, the main conflict resolution method was based on recency, with preference being given to instantiations that matched against more recent elements. Rychener used this bias to ensure that subgoals were addressed in the desired order. Second, PSNLST dynamically reordered productions according to the number of times they had been placed on the candidate match list. This led the architecture to prefer rules that had recently shown promise, and had a quite different flavor from the static ordering schemes that had prevailed in earlier languages.

The history of production systems took another significant turn when Forgy & McDermott (1976) developed the first version of OPS. This language was the vehicle of the Instructable Production System project at Carnegie–Mellon University, led by Allen Newell. The most important feature of OPS was its pattern matcher, which allowed it to efficiently compute the matched instantiations of all productions. This in turn made possible experiments with different conflict resolution strategies, and made the notion of refraction tractable. OPS also assumed a working memory of unlimited size, with the associated the-

[5] However, PAS aided the cognitive simulation effort indirectly, since Waterman & Newell (1972) used it to construct a semi-automatic system for analyzing verbal protocols.

oretical claim that human memory only *appeared* limited due to a recency-based conflict resolution scheme (similar to that used by PSNLST).

Forgy's initial version of OPS was followed by OPS2 (1977) and OPS4 (1979a), which were minor variations implemented in different programming languages. By OPS4, the basic conflict resolution strategies had stabilized, with refraction given first priority, followed by recency of the matched elements and specificity of the rules. Rychener's (1980) OPS3 was a descendant of OPS2 that stored elements in terms of attribute-value pairs in an effort to provide greater representational flexibility.

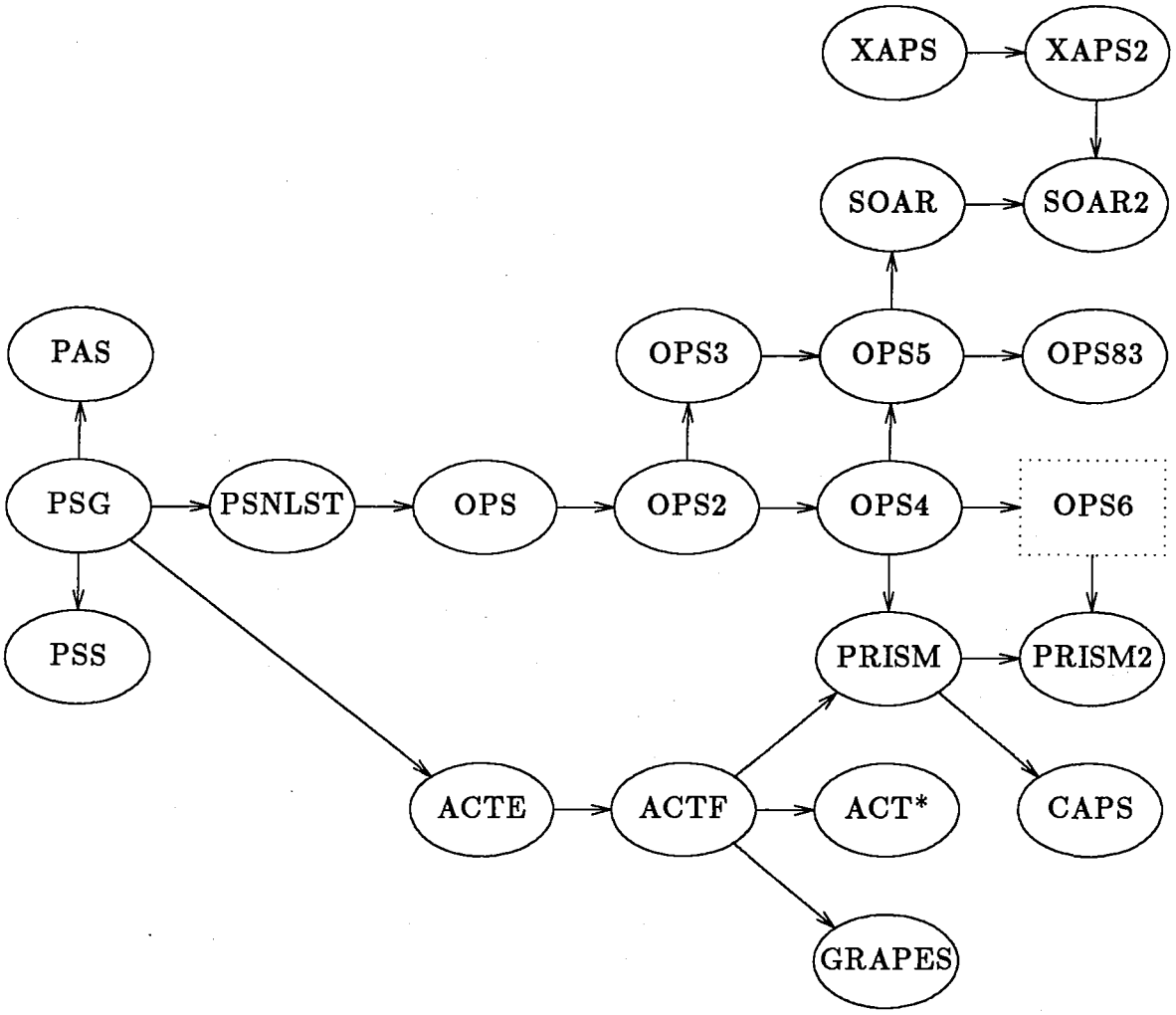

Figure 1. Development of production system architectures.

The next incarnation, OPS5 (Forgy, 1981), also used an attribute-value scheme, but in a much less flexible manner than OPS3 did. However, the main goal of OPS5 was efficiency, and along this dimension it succeeded admirably. Over the years, it has become one of the most widely used AI languages for implementing expert systems, but it was never intended as a serious model of the human cognitive architecture. We direct the

reader to Brownston, Farrell, Kant, & Martin (1985) for an excellent tutorial in OPS5 programming techniques. More recently, Forgy (1984) has developed OPS83, a production system language with many of the features of PASCAL, making it an even less plausible cognitive model than was OPS5 (though again, this was not its goal).

Although never implemented, OPS6 (Rosenbloom & Forgy, personal communication) included a number of ambitious ideas. For instance, both lists and sets were to be supported as basic data structures in both working memory elements and productions. In addition, multiple working memories and production memories were to be allowed, each having its own sub-architectural characteristics. This flexibility was in direct contrast to the restrictions imposed for the sake of efficiency in OPS5.

In parallel with the work at C–MU, Anderson and his colleagues at Yale were developing ACTE (1976), a production system framework that was intended to model human behavior. This architecture included two declarative repositories – a long-term memory and an "active" memory that was a subset of the first. Productions matched only against elements in active memory, but these could be retrieved from long-term memory through a process of spreading activation. Conflict resolution was tied to the strength and specificity of rules, and to the activation levels of the matched elements; moreover, the final selection of instantiations was probabilistic. Its successor was ACTF (Anderson, Kline & Beasley, 1978), an architecture that included mechanisms for learning through generalization, discrimination, and strengthening of rules. Methods for learning by composition and proceduralization were added later (Neves & Anderson, 1981).

When Anderson moved to Carnegie–Mellon University in 1979, a new generation of languages began to evolve that incorporated Forgy's matcher. Two architectures – ACTG and ACTH – existed only in transitory form, but these were followed by GRAPES (Sauers & Farrell, 1982), a production system language that placed goals in a special memory. Methods for learning by composition and proceduralization were also included in GRAPES, and were closely linked to the processing of goal structures.

Concurrently with the GRAPES effort, Anderson (1983) developed ACT*, apparently the final installment in the ACT series of architectures. This framework employed the notion of excitation and inhibition among the conditions of various productions, and implemented a partial matching scheme based on a variant of Forgy's matcher. ACT* also employed three forms of declarative representation – list structures (like those used in most production systems), spatial structures, and temporally ordered lists. The architecture also supported a variety of learning methods carried over from ACTF, though the notion of spreading activation differed considerably from earlier versions.

The short-lived ACTG was developed by Langley, but a collaboration with Neches led to PRISM (Langley & Neches, 1981), a production system language that supported a *class* of cognitive architectures in the same spirit as PSG. PRISM incorporated some features of OPS4, ACTF, and HPM (Neches, this volume), including two declarative memories, mechanisms for spreading activation, and methods for learning by discrimination, generalization, and strengthening. Since it employed the OPS4 matcher, the language provided considerable flexibility in conflict resolution. Based on the PRISM code, Thibadeau (1982)

developed CAPS, an activation-based architecture that employed a notion of thresholds and allowed rules to fire in parallel.

PRISM's major limitation was the large number of unstructured parameters used to specify a given architecture. In response, the researchers developed PRISM2 (Langley, Ohlsson, Thibadeau & Walter, 1984), which organized architectures around a set of schemas, each with a few associated parameters. This language was strongly influenced by the Rosenbloom & Forgy OPS6 design, and allowed arbitrary sets of declarative and production memories, each with its own characteristics (such as decay functions and conflict resolution schemes).

Rosenbloom's interest in motor behavior led to XAPS (1979) and XAPS2 (this volume), architectures that allowed parallel application of productions. The first of these was activation-based, while the distinguishing feature of the second was its method for learning by chunking. Laird (1983) developed SOAR, an architecture that combined the production system framework with Newell's (1980) problem space hypothesis. Finally, Laird, Rosenbloom & Newell (1984) reported SOAR2, an architecture that combined the problem solving features of SOAR with the chunking abilities of XAPS2.

We have attempted to summarize the developmental influences among these architectures in Figure 1. For instance, the sequence of OPS languages descended directly from one another, though OPS3 and OPS4 should be viewed as co-existing rather than as parent and child. ACTF and OPS4 were the main influences on the initial version of PRISM, while PRISM2 derived from PRISM and the OPS6 design. We have not attempted to give an exhaustive treatment here. However, the architectures we have reviewed seem (to us) to be the mainstream of production system research, and most of the architectures described in this volume can be traced back to one of the frameworks we have described.

The Nature of Learning

The term "learning" is similar to the term "art", in that it is very difficult to generate a satisfactory definition of either term.[6] Learning is clearly a multi-faceted phenomenon that covers a variety of quite different behaviors. Perhaps the most commonly accepted "definition" is that learning is "the improvement of performance over time". However, other researchers view learning as "increased understanding", while still others view learning as involving "data compression" or the summarization of experience.

Examples of Learning

Rather than attempt to define such an ambiguous term, let us consider some examples of learning "tasks" that have been discussed in the literature of cognitive psychology and artificial intelligence. The most common of these is the task of "learning from examples", in which one is presented with positive and negative instances of some concept, and attempts to formulate some general description that covers the positive instances but none of the negative instances. For this class of tasks, improvement consists of increasing one's ability

[6] Bundy (1984) discusses the difficulties of defining "learning" in some detail.

to correctly predict exemplars and non-exemplars of the concept. Chapter 3 gives a fuller treatment of this problem, which has a long history within both psychology and AI. In many ways, this is the simplest of the learning tasks that has been studied (Winston, 1975; Hayes-Roth & McDermott, 1978; Anderson & Kline, 1979; Michalski, 1980).

A more complex task is that of "learning search heuristics", in which one is presented with a set of problems that require search in order to find the solution. After suitable practice on these tasks, the learner is expected to acquire heuristics that reduce or eliminate the search he must carry out in solving the problems. This task has received considerable attention within AI in recent years, and is also discussed in greater detail in Chapters 2 and 3. The vast majority of work in this area has been within the production system framework (Brazdil, 1978; Neves, 1978; Langley, 1983; Mitchell, Utgoff & Banerji, 1983), though only a few researchers have attempted to model human learning in this domain (Anzai, 1978; Ohlsson, 1983).

In the heuristics learning task, the notion of improvement involves a reduction in search. On another class of procedural learning tasks, improvement consists of reduction in the amount of time required to carry out an algorithmic procedure. Chapters 4, 5, and 7 all focus on how such speedup effects can result from practice. In general, the mechanisms proposed to account for speedup have been quite different from those posited to account for reduced search (Lewis, 1978; Neches, 1981a; Neves & Anderson, 1981). However, Laird, Rosenbloom & Newell (1984) have recently outlined a production system theory that accounts for both using the same mechanism.

The task of language acquisition has also received considerable attention, though most work in this area has focused on the subproblem of grammar learning (Hedrick, 1976; Berwick, 1979; Anderson, 1981; Langley, 1982). In this task, the learner hears grammatical sentences from the language to be acquired, together with the meanings of those sentences.[7] The goal is to generate rules for mapping sentences onto their meanings, or meanings onto grammatical sentences. In the grammar learning task, improvement involves generating ever closer approximations to the adult grammar. This task is described in more detail in Chapter 3.

The final class of tasks we will consider (though we have not attempted to be exhaustive) is that of cognitive development. Obviously, this term covers considerable ground, but in this context we assume a fairly specific meaning. Cognitive developmental researchers have studied children's behavior on a variety of tasks, many originally invented by Piaget. There is considerable evidence that children progress through identifiable stages on each of these tasks, and one would like some theory to account for this progression. In this case, improvement consists of changes that lead toward adult behavior on a given task. Relatively few learning models have been proposed to account for these data, but we will see two of them in Chapters 3 and 8.

In addition to accounting for improvement, models of human learning should also account for side effects of the learning process. For example, one effect of speedup seems

[7] Typically, these are presented as sentence-meaning pairs, though it is theoretically assumed that the child must infer the meanings from context.

to be a lack of flexibility in the resulting procedure. Neves & Anderson (1981) have shown that such phenomena arise naturally from their composition model of speedup, leading to Einstellung in problem solving behavior. The list of such successful predictions is small, but at this stage in the development of our science, even theories that account for the fact of improvement must be considered major advances.

The reader may have noted that we have focused on learning from experience, as opposed to learning from instruction or learning by deduction. Similarly, we have focused on the acquisition of procedures, rather than the acquisition of declarative knowledge. This reflects a bias of the editors, and this bias is reflected in the following chapters (the chapter by Wallace, Klahr, and Bluff is the exception, addressing non-procedural issues). We will not attempt to justify this bias, for certainly non-procedural, non-experiential learning is equally a part of the human condition and must ultimately be included in any integrated model of human behavior. However, we should note that our emphasis on procedural, experience-based learning reflects current trends in cognitive science and AI, so we feel our bias is representative of these fields.

Components of Learning

Now that we have examined some examples of learning tasks, let us turn to some common features of these tasks. In particular, within any task that involves learning from experience, one can identify four component problems that must be addressed.[8] These components are:

- *Aggregation.* The learner must identify the basic objects that constitute the instances from which he will learn. In other words, he must identify the *part-of* relations, or determine the appropriate *chunks.*
- *Clustering.* The learner must identify which objects or events should be grouped together into a class. In other words, he must identify the *instance-of* relations, or generate an *extensional* definition of the concept.
- *Characterization* The learner must formulate some general description or hypothesis that characterizes instances of the concept. In other words, he must generate an *intensional* definition of the concept.[9]
- *Storage/Indexing.* The learner must store the characterization in some manner that lets him retrieve it, as well as make use of the retrieved knowledge.

We would argue that any system that learns from experience must address each of these four issues, even if the responses to some are degenerate. In fact, we will see that many of the traditional learning tasks allow one to ignore some of the components, making them idealizations of the general task of learning from experience.

[8] Fisher & Langley (1985) have used some of these components to analyze methods for "conceptual clustering", while Easterlin & Langley (1985) have examined their role in concept formation.

[9] This is often called the "generalization" problem, but since this word has an ambiguous meaning, we will avoid its use here.

For instance, the task of learning from examples can be viewed as a degenerate case of the general learning task, in that the tutor solves the aggregation and clustering problems by providing the learner with positive and negative instances of the concept to be learned. In addition, since only one (or at most a few) concept(s) must be acquired, there is no need to index them in any sophisticated manner – a simple list is quite sufficient. Thus, the task of learning from examples can be viewed as "distilled" characterization, since this is the only component of learning that must be addressed. This simplification has proven quite useful to learning researchers, and many of the characterization methods that were initially developed for the task of learning from examples have been successfully transferred to more complex problems.

Similarly, the task of heuristics learning ignores the issue of aggregation, assuming that individual problem states constitute the objects upon which learning should be based.[10] However, methods for heuristics learning must directly respond to the clustering issue, since no tutor is available to identify positive and negative instances. In fact, within the framework of learning search heuristics, the clustering problem is identical to the well-known problem of credit and blame assignment. Methods for heuristics learning must also address the characterization problem, since one must generate some general description of the situations under which each operator should be applied. Finally, the storage issue has generally been ignored in research on heuristics learning, since relatively small numbers of operators have been involved.

Many of the models of speedup involve some form of chunking, and this approach can be viewed as another variant on the general task of learning from experience. In this case, the structure to be learned is some sequential or spatial configuration of perceptions or actions. Unlike the tasks of learning from examples and heuristics learning, chunking tasks force one to directly address the *aggregation* issue, since one must decide which components to include as parts of the higher level structure. However, in most chunking methods, the *characterization* problem is made trivial, since new rules are based directly on existing rules, for which the level of generality is already known. Also, even in methods that address issues of characterization, chunks are based on single instances, so that the *clustering* problem is also bypassed.

Finally, the grammar learning task can be viewed a fourth variant on the general problem of learning from experience. Like the chunking task, it forces one to respond to the problem of *aggregation*, since it must form sequential structures such as "noun phrase" and "verb phrase". Like the heuristics learning task, it requires one to address the *clustering* problem, since it must group words into disjunctive classes like "noun" and "verb" without the aid of a tutor. It also forces one to deal with characterization issues, though in some approaches, the set of chunks in which a class like "noun" occurs can be viewed *characterization* of that class. Storage is also a significant issue in grammar learning, since many rules may be required to summarize even a small grammar.

[10] As we will see in Chapter 3, this is not completely true, but it has been true of the majority of work on learning search heuristics.

It is interesting to note that production system models of learning sometimes provide their own answer to the storage/indexing problem, since the conditions of productions can be efficiently stored in a discrimination network that both reduces space requirements and speeds up the matching process. Forgy (1982) has described this storage scheme in some detail and many of the models described in this book rely on Forgy's approach. Thus, one can argue that within the production system framework, only the issues of aggregation, clustering, and characterization must be addressed. Although some readers may disagree with this assumption, most of the chapters in this volume adopt it either implicitly or explicitly.

Mechanisms of Learning

In the preceding sections, we considered the learning *phenomena* that require explanation, followed by the *components* of learning that any model must address. In this section, we consider various *mechanisms* that have been proposed to account for the learning process, both in the following chapters and elsewhere in the literature.

Within the production system framework, the nature of the recognize-act cycle constrains the points at which learning can have an effect. There are three such choice-points; a production system's repertoire of behaviors can be changed by affecting the outcome of:

- the process of matching productions
- the process of conflict resolution
- the process of applying productions

Let us examine different mechanisms that have been proposed in terms of the manner in which they affect the recognize-act cycle.

The most obvious way to affect the set of applicable productions found by the matching process is to add new productions to the set. However, this makes certain assumptions: matching must either be exhaustive, or the new production must be added in such a way that guarantees it will be considered by the matcher. Waterman's (1970, 1975) early work took the latter approach, adding new rules above those productions they were intended to mask (since rules were matched in a linear order). The work on stage models of development (Young, 1976) employed similar methods. After Forgy (1979b) presented an efficient method for computing all matched instantiations, the exhaustive scheme became more popular.

The earliest approach to creating new rules employed a production-building command in the action side of productions. This function took an arbitrary condition-action form as its argument, and whenever the rule containing it was applied, it would instantiate the form and add a new rule to production memory. Of course, simple instantiations of a general form are not very interesting, but one could use other productions to *construct* more useful rules which were then passed to the production-building rule. The OPS family of architectures called this function *build*, while the ACT family used the term *designate*. Although this method was quite popular in the early work on adaptive productions, it has now been largely replaced by other methods.

Another way to affect the set of matched productions is to modify the conditions of existing rules, or to construct variants on existing rules with slightly different condition sides.[11] In practice, most researchers have opted for the latter approach, sometimes in order to model the incremental nature of human learning, and other times responding to complexities in the learning task, such as the presence of noise. In such approaches, the generation of variants must be combined with methods for modifying the conflict resolution process, so that better variants eventually come to mask the rules from which they were generated.

Two obvious methods present themselves for modifying the conditions of productions. Although these mechanisms had been present in the AI literature for some time, Anderson, Kline & Beasley (1978) were the first to use them in models of human learning, labeling them with the terms *generalization* and *discrimination.* The first process involves the creation of a new rule (or modifying an existing one) so that it is *more* general than an existing rule, while retaining the same actions. The term "generalization" has also been used to denote *any* process for moving from data to some general rule or hypothesis, but Anderson intended a much more specific sense. The second process of discrimination involves the creation of a new rule (or modifying an existing one) so that it is *less* general than an existing rule, while retaining the same actions. In their effects, the two mechanisms lead to opposite results, though in most models they are not inverses in terms of the conditions under which they are evoked.

Within production system models, there are three basic ways to form more general or specific rules, each corresponding to a different notion of generality. First, one can add or delete conditions from the left-hand side of a production. The former generates a more specific rule, since it will match in fewer situations, while the latter gives a more general rule. The second method involves replacing variables with constant terms or vice versa. Changing variables to constants reduces generality, while changing constants to variables increases generality. The final method revolves around the notion of class variables or is-a hierarchies. For example, one may know that both dogs and cats are mammals, and that both mammals and birds are vertebrates. Replacing a term from this hierarchy with one below it in the hierarchy decreases generality, while the inverse operation increases generality.

These techniques have been used in programs modeling behavior on concept acquisition experiments (Anderson & Kline, 1979), language comprehension and production at various age levels (Langley, 1982; Anderson, 1981), geometry theorem proving (Anderson, Greeno, Kline & Neves, 1981), and various puzzle-solving tasks (Langley, 1982). Although we will not elaborate on these learning mechanisms here, Langley (this volume) describes them in considerable detail.

---

[11] For this to be different from simply adding a new production, another constraint is required: the conditions must be composed of simple parts, rather than invocations to very powerful predicates. If such predicates were allowed, one might change the matched set by redefining some predicate in an arbitrary manner. Although theoretically possible, this scheme would make the learning process hopelessly complex.

Note that methods like discrimination and generalization directly respond to the characterization issue we described in the last section. Both methods require instances that have been clustered into some class, and both attempt to generate some general description of those classes based on the observed instances. As one might expect, these methods were first proposed by researchers working on the task learning from examples (Winston, 1975; Hayes-Roth & McDermott, 1978; Mitchell, 1982; Brazdil, 1978).

Once a set of matching rule instantiations have been found, a production system architecture still must make some determination about which instantiation(s) in that set will be executed. Thus, conflict resolution offers another decision point in the recognize-act cycle where the behavior of the system can be affected. This turns out to be particularly important because many models of human learning attempt to model its incremental nature, assuming that learning involves the construction of successively closer approximations to correct knowledge over a series of experiences.

The knowledge represented in new production rules are essentially hypotheses about the correct rules. A learning system must maintain a balance between the need for feedback obtained by trying new productions and the need for stable performance obtained by relying on those productions that have proved themselves successful. This means that a learning system must distinguish between rule *applicability* and rule *desirability*, and be able to alter its selections as it discovers more about desirability. Production systems have embodied a number of schemes for performing conflict resolution, ranging from simple fixed orderings on the rules in PSG (Newell & McDermott, 1975) and PAS (Waterman, 1975), to various forms of weights or "strengths" (Anderson, 1976; Langley, this volume), to complex schemes that are not uniform across the entire set of productions as in HPM (Neches, this volume). Combined with certain timing assumptions, these schemes can be used to predict speedup effects as well as affecting more global behavior.

Any production system model of learning must take into account the parameters used during conflict resolution. This concern goes beyond learning models, since any system with multiple context-sensitive knowledge sources may have overlapping knowledge items that conflict.[12] However, the problems of conflict resolution are exacerbated in a learning system because the knowledge items being resolved may change over time, and because new items must be added with consideration for how conflicts with existing items will be handled. Later in the chapter, we will return to the constraints that learning imposes on acceptable conflict resolution mechanisms.

After conflicts have been dealt with, the instantiations(s) selected for execution are applied. Although the opportunities arising at this point largely parallel those involved in matching, there is little parallel in the methods that have been proposed. For instance, one can imagine methods that add or delete the actions of existing rules, but we know of only one such suggestion in the literature (Anderson, 1983).

However, three additional learning mechanisms have been proposed that lead to rules with new conditions *and* actions. The first of these is known as *composition*, and was orig-

[12] The traditional example used to illustrate this point contrasts the two pieces of decision-making advice – "Look before you leap" and "He who hesitates is lost".

inally proposed by Lewis (1978) to account for speedup as the result of practice. Basically, this method combines two or more rules into a new rule with the conditions and actions of the component rules. However, conditions that are guaranteed to be met by one of the actions are not included. For instance, if we compose the rules (A B → C D) and (D E → F), the rule (A B E → C D F) would result. Of course, the process is not quite this simple; most composition methods are based on instantiations of productions rather than the rules themselves, and one must take variable bindings into account in generating the new rule. The presence of negated conditions also complicates matters.

Note that composition is a form of chunking, and thus is one response to the aggregation problem we discussed earlier. However, the combination procedure outlined above is not sufficient; this must be combined with some theory about the conditions under which such combinations occur. Some theories of composition (Lewis, 1978; Neves & Anderson, 1981) assume that composition occurs whenever two rules apply in sequence, while others (Anderson, 1983) posit that composition transpires whenever some goal is achieved. Naturally, different conditions for chunking lead to radically different forms of learning. The chapter by Lewis describes the composition process in more detail.

Newell & Rosenbloom (1981) have proposed another response to the aggregation issue in their theory of learning by chunking, showing that the learning curves predicted by their model are quite similar to those observed in a broad range of learning tasks. The chapter by Rosenbloom and Newell (this volume) presents results from this approach to modeling practice effects.

Yet another mechanism for creating new rules has been called *proceduralization* (Neves & Anderson, 1981). This involves constructing a very specific version of some general rule, based on some instantiation of the rule that has been applied. Ohlsson (this volume) has used a similar mechanism to model learning on transitive reasoning tasks. In some ways, this method can be viewed as a form of discrimination learning, since it generates more specific variants of an existing rule. However, the conditions for application tend to be quite different, and the use to which these methods have been put have quite different flavors. For instance, discrimination has been used almost entirely to account for reducing search or eliminating errors, while proceduralization has been used to account for speedup effects and automatization. But perhaps these are simply different names for the results of the same underlying phenomena.

Additional Learning Mechanisms

Langley, Neches, Neves & Anzai (1980) have argued that self-modifying systems must address two related problems: inducing *correct* rules for when to perform the various actions available to the system, and developing *interesting* new actions to perform. However, most of the models that have been developed in recent years have focused on the first of these issues, and some researchers (e.g., Anderson, 1983) have asserted that mechanisms such as composition, generalization, and discrimination are sufficient to account for all learning.

Nevertheless, evidence is starting to build up that these processes, although apparently necessary components of a computational learning theory, are by no means sufficient. The

evidence for this comes from a number of recent studies which have tried to characterize differences between the strategies employed by experts and novices. For example, Lewis (1981) has documented differences between expert and novice solution methods for algebra expressions, and shown that the differences could not have been produced by composition. Lewis' argument consists of a demonstration that the procedures produced by a process of composition would not apply correctly in all cases. In order to ensure that a new procedure would work correctly, additional rules must be produced by some process other than composition.

Another example of complex expert strategies appears in Hunter's (1968) analysis of the procedures employed by a "mental calculator", a subject with highly exceptional skills at mental arithmetic. There appear to be a number of aspects to the subject's special abilities. Some, such as his large collection of number facts, might be explained in terms of mechanisms like the Neves & Anderson (1981) "knowledge compilation" model. However, there are many aspects of the subject's performance for which it is very difficult to see how syntactic learning mechanisms could have produced the observed results.

For instance, Hunter found that his subject's superior ability to mentally solve large multiplication problems was due to a procedure that performed the component multiplications in left-to-right order while keeping a running total of the intermediate products. This contrasts to the traditional pencil-and-paper algorithm in which columns are multiplied right-to-left, with the sub-products written down and totaled afterwards in order to compute the product. The left-to-right procedure, which drastically reduced the working memory demands of any given problem, requires a massive re-organization of the control structure for the traditional multiplication procedure.

The re-organization involves much more than refinements in the rules governing when sub-operations are performed. Such refinements could presumably be produced by generalization and discrimination mechanisms. However, producing this new procedure requires the introduction of new operations (or at least new goal structures), such as those involved in keeping a running total of the sub-products. Those new operations, and the control structure governing the sequence of their execution, require the introduction of novel elements or goals – something that generalization, discrimination, and composition are clearly not able to do.

Similar conclusions can be drawn from studies on expert/novice differences in physics problem solving (Simon & Simon, 1978; Larkin, 1981). A general observation in those studies is that experts rely on working-forward strategies while novices are much more inclined to use means-ends analysis. Generally, the mechanism used to explain this relies on a method first developed by Anzai called "block-elimination" (Anzai & Simon, 1979). In this method, one remembers a state S in which some desired action (operator) A cannot be executed because its preconditions are not met. If heuristic search later generates another action B that eliminates the blockage (enabling A to be applied), then a new rule is constructed. This rule has the form "If you are in state S and you want to apply action A, then try to apply action B." In other words, this approach leads to new subgoals and to rules which propose when they should be generated.

The examples presented thus far provide motivation for seeking learning mechanisms beyond composition, proceduralization, generalization, and discrimination. However, our argument has rested on cases in which no evidence was available about the intermediate forms of the acquired procedures. There are very few studies in which learning sequences, and the intermediate procedures produced within them, have been directly observed. Fortunately, a similar picture emerges from two studies in which those observations could be made.

In the first of these studies, Neches (1981b) traced procedure development in the command sequences issued by an expert user of a computer graphics editing system. In doing this, he found a number of changes that involving re-ordering operations and re-planning procedure segments on the basis of efficiency considerations. In the second study, Anzai & Simon (1979) examined a subject solving and re-solving a five-disk Tower of Hanoi puzzle. They found a number of changes in procedure that seemed inconsistent with strict composition/generalization/discrimination models. These included eliminating moves that produced returns to previously visited problem states, establishing subgoals to perform actions that eliminated barriers to desired actions, and transforming partially specified goals (such as moving a disk off a peg) into fully-specified goals (such as moving the disk from the peg to a specific other peg).

A key observation about the above examples is that the learning appears to involve reasoning on the basis of knowledge about the structure of procedures in general, and the semantics of a given procedure in particular. In each of the examples we have considered, procedures were modified through the construction of novel elements rather than through simple deletions, additions, or combinations of existing elements. This leads us to believe there exist important aspects of learning that involve the use of both general and domain-specific knowledge about procedures. The chapter by Neches (this volume) examines this form of learning in more detail.

Implications of Learning for Production System Architectures

In the preceding sections, we have been concerned with the mapping between learning phenomena (viewed from a psychological perspective) and explanations of these phenomena in terms of production systems. Attempts to build production system-based learning models have led, among other things, to a much better understanding of what does and does not work in production system architectures. In this section, we try to summarize the lessons that that have been learned for three major aspects of production systems.

Pattern Matching

In practice, languages for expressing conditions on productions generally turn out to be fairly simple. In particular, they generally rely on pattern-matching rather than the testing of arbitrary predicates. That is, the language of condition-sides describes abstract configurations of data elements. A production's conditions are said to be satisfied when data appears in working memory that instantiates the described configuration in a consistent manner. This means that the condition side specifies a pattern consisting of

constants and variables, and the data in working memory matches that pattern by utilizing the same constants in the same places as they appeared in the pattern, with constants that correspond to variables in the pattern appearing consistently in the same places as those variables.

The description in the preceding paragraph may seem both obvious and redundant, given our discussion of production systems. However, note that rule-based systems need not operate within a pattern-matching paradigm. In principle, the condition sides of production rules could be arbitrarily complex and have arbitrarily powerful predicates.[13] From a logical standpoint, any class of expressions that evaluated to true or false (applicable or not-applicable) could serve to specify production conditions. It is worthwhile to examine the advantages of the pattern-matching paradigm. We will argue that there are three constraints on production systems which have emerged from learning research, and which the pattern-matching paradigm serves to satisfy:

- *Efficiency and adaptivity* – Since the grain size of productions is relatively small (i.e., little happens with each production firing), and since humans are real-time systems that must make timely responses to external events, it is important to iterate through the recognize-act cycle as quickly as possible.
- *Even granularity* – Learning mechanisms often depend on taking fragments of different existing rules, making inferences about the contribution of those fragments to the "success" or "failure" of the rule, and modifying rules by adding or deleting those fragments. Therefore, it is important that those fragments all represent small and approximately equal units of knowledge. If a fragment represents too large a decision, then the reasoning that went into that decision is not accessible to learning mechanisms. This means that that reasoning cannot be re-used elsewhere if it is correct, and cannot be corrected if it is not.
- *Analyzability* – As just mentioned, learning mechanisms operate by manipulating fragments of rules (or instantiations of rules), either to construct new rules or to "edit" existing rules. This can only occur if the mechanisms can predict the effect their manipulations will have on behavior, and this in turn requires that there be consistent principles about the nature of conditions and the effects of altering them. Put another way, for learning mechanisms to operate in a domain-independent way (i.e., without built-in knowledge about the domain in which learning is taking place), these mechanisms must operate on the structure or syntax of productions, rather than upon their content or semantics.

To understand the implications of these constraints, let us first consider why they rule out the extreme opposite of pattern matching: using the connectives of first-order predicate calculus to combine predicates implemented as functions that return a truth value. In a rule-based system that allowed such conditions, one could generate rules with condition sides that required extremely long (if not infinite) amounts of time to test, thus violating the efficiency constraint. It would be possible to (in fact, it would be difficult *not* to) have a set of predicates that differed widely in their complexity, sophistication, and generality, thus

[13] For an extreme example of this approach, see Lenat (1977).

discussion of these issues, see McDermott & Forgy (1978).) The production system must give enough priority to the results of its own recent activity to focus upon a goal and carry it through to completion (as opposed to oscillating between many different goals while only slowly making progress on any given goal). At the same time, humans must interact with an external environment, and this requires that production system models of human behavior be able to shift attention in response to events in the outside world.

In order to meet these demands, production systems generally employ a hybrid approach, in which the set of applicable productions is whittled down by applying a sequence of conflict resolution policies, rather than a single all-encompassing policy. OPS2 (Forgy & McDermott, 1977) is an example of this approach. The OPS3 conflict resolution scheme operates in five successive stages:

- *Refraction*: Once executed, an instantiation of a given production may not be executed again (until one of the matched elements has been deleted and readded to working memory);
- *Recency*: Production instantiations matching recently asserted data should be selected over those matching older data;
- *Specificity*: Instantiations of productions with the greatest number of condition elements and constant terms should be selected;
- *Production recency*: Instantiations of recently created productions should be selected over those from older productions;
- *Random selection*: If the above rules were not sufficient to produce a unique selection, then a production instantiation should be chosen at random.

These policies were intended to satisfy a number of concerns. Refraction ensured that the system would move on to new tasks and would not tend to enter infinite loops. Recency ensured that the system would not tend to oscillate between goals, but would focus on the most recent one, attending to that goal until it was accomplished. At the same time, this policy still allowed for switching attention in response to external events, since those also would entail new data in working memory.

The third criterion, specificity, attempted to ensure that the "best" production would be selected. Since rules containing more conditions and more constant terms tend to be more specific than those with fewer conditions or constants, and therefore the one most closely tailored to fit the current situation, they should be most appropriate. The criterion of production recency was intended to favor the results of learning. Much like the notions behind production ordering, the assumption was that new rules were likely to be improvements over old rules, and therefore should be given an opportunity to mask the prior ones. The final policy of random selection was intended to ensure that the system would only fire one production per cycle; in parallel systems that allow more than one production to be executed in a cycle, it is possible to execute productions that are at cross-purposes.

Policies like those in the OPS family of languages have generally been successful at controlling production systems that do not change greatly. They have been used in practical applications, such as McDermott's (1982) R1 expert system, involving large numbers of productions and calling for a high degree of efficiency. However, though they are clearly an improvement, such policies have not proved completely satisfactory where learning is concerned. Fortunately, Forgy's (1979b) work on matching provides a method for efficiently computing all matched rules. This in turn provides considerable flexibility in the conflict resolution process, so that one is not locked into a particular set of decision criteria.

One early target of dissatisfaction concerned criteria that favored newly acquired rules over older rules. Such policies ignored the need for learning mechanisms that relied on incremental refinement of rules. Any policy that always favors new productions ignores the possibility that intermediate stages of learning might produce errorful rules. Furthermore, such a policy provides no help in selecting between multiple variants of an existing rule which are generated by some learning mechanisms. Langley's work on the AMBER and SAGE systems (this volume) illustrates both the problems and the general approach taken towards solving them in many recent production system architectures.

The basic idea was to introduce a notion of production *strength* as a factor in conflict resolution. The strength (or weight) of a production is a parameter that is adjusted to indicate the system's current confidence in the correctness and/or usefulness of that rule. There have been a number of different approaches taken in various production system architectures toward making use of this general concept. In some variants, the strength or weight of productions are subject to a threshold; rules below the threshold simply are not considered. In other variants, the strength is a positive factor in selection; "stronger" rules are preferred over weaker ones. In still other variants, the notion of strength is used to make selection less deterministic; strength is treated as a measure of the probability that a rule will fire given that its conditions are satisfied.

Although these variants all make different statements about the precise usage of production strength, the particular mechanisms all serve a common purpose – enabling a phase of evaluation and testing for proposed rules, rather than forcing an all-or-nothing decision about them. To this end, mechanisms for affecting the strength of a rule are needed. Some researchers have postulated automatic mechanisms. For example, in Langley's AMBER and SAGE systems (this volume), a rule is strengthened each time a learning mechanism rediscovers it; thus, more weight is given to rules that have been learned a number of times, even if they have been created by different learning methods.

Although the notion of production strength is probably the major change in production system architectures that has resulted from research on learning, other problems have also received attention. In particular, Neches (1981b) has argued that a uniform conflict resolution scheme places an excessive burden on learning mechanisms. (By a uniform conflict resolution scheme, we mean a set of conflict resolution policies that are applied consistently to all applicable productions with no special exceptions.) The argument, which is only briefly touched upon in his chapter in this volume, is that a uniform conflict resolution scheme forces several different kinds of knowledge to be confounded in the conditions of productions. Three kinds are particularly important:

- Conditions that define the context in which the production is applicable – in other words, knowledge about the circumstances in which it is meaningful to execute the production;
- Conditions that enable processing on the action side of a rule, usually by causing values to be assigned to variables that are referenced on the action side – in other words, knowledge about prerequisite information for operations on the right-hand side;
- Conditions that serve to affect flow of control by causing otherwise applicable productions to be inapplicable or by affecting the ranking of applicable productions during the selection process – in other words, heuristic knowledge about desired sequencing of productions.

In response to these observations, Neches has proposed having different conflict resolution associated with different *classes* of rules, with each class being responsible for different aspects of behavior.

Working Memory

All modern notions of the human information processing architecture, although differing in their details, agree on the importance of distinguishing two aspects of memory. On the one hand, there seems to be a limited capacity working memory that changes rapidly over time. Working memory plays the same role for humans as input/output buffers and processing registers play for computers. On the other hand, there is also a large capacity, long term memory that is much more stable. We have already seen how production system models instantiate this distinction.

The production system for subtraction that we presented earlier was quite simple, requiring only a few types of working memory elements. However, this was a performance model, and learning systems require more information than is strictly necessary for performance.[14] This extra information must be retained somewhere, and the most likely place is working memory. Thus, a focus on learning imposes constraints on models of working memory that would not arise from a concern with performance alone.

The state of the art is not sufficiently advanced that we can exhaustively list all of the different kinds of information used by learning mechanisms, but we can extract a partial list from the mechanisms discussed elsewhere in this chapter. For example, the minimum information requirements of composition methods are behavioral records (in the form of at least two production instantiations), along with temporal information (in the form of

[14] For example, Neches (1981b) compares a production system for learning an addition procedure (also described in his chapter in this volume) with another production system that was adequate for performing the task, but which did not carry the information needed for his learning mechanisms to operate. Although the number of productions contained in the two systems was approximately the same, the "complexity" of the productions needed in the learning system was almost three times greater for both of two different measures. (Complexity was measured by the number of symbols and the number of propositions appearing in the production.)

the sequential ordering between the two instantiations).

Generalization and discrimination methods require more information, including knowledge of the success/failure of rules and the context in which that success or failure occurred. In simple tasks such as learning from examples, immediate feedback is provided to the learner. However, more complex learning tasks require the system to *infer* the success or failure of individual rules; Minsky (1963) has called this the *credit assignment* problem. Moreover, since credit or blame may not be assigned to a rule until long after it has applied, the context of that application (generally the state of working memory at that point) must be stored for later retrieval.

For example, when generalization and discrimination methods are used to determine heuristic conditions for problem-solving operators (Sleeman, Langley & Mitchell, 1982), the learning system must retain information about its search towards a problem solution. After the problem is solved by weak search methods, feedback is obtained by asking whether or not actions that were tried were on the final solution path. Credit assignment is performed by attributing success to the productions that suggested actions on the solution path, and by attributing failure to productions that suggested actions leading off the final solution path. In these more sophisticated models, information must be retained about the sequential ordering of previous knowledge states, as well as the states themselves.

Similar themes can be found in Anzai's work on "learning by doing", which is reviewed as part of his chapter in this book. Anzai models a number of complementary processes that contribute to the development of procedures for solving problems that can initially only be solved by general search methods. His view of learning involves several phases. In the first phase, a system learns to narrow the search space by avoiding unproductive branches in the search tree ("bad moves"). The key to this phase is the use of heuristics for identifying bad branches, such as noting action sequences which return a problem solution to a previous state. This first phase involves retaining much the same sort of knowledge as required by generalization and discrimination.

In Anzai's second phase of learning, a system attempts to infer subgoals ("good moves"). This depends on noting actions that cannot be carried out when they are first proposed because some preconditions are violated, but which are applied later, after another action has satisfied those preconditions. This second phase of learning requires information that goes beyond that needed for the first phase, since it must retain in memory both unsatisfied goals and the reasons they were not satisfied. More generally, the kind of information being retained has to do with planning, that is, the consideration of actions separate from the performance of those actions.

Anzai's later phases of learning involve acquiring associations between action sequences and the subgoals learned in the second phase, as well as grouping sequences into useful units. Thus, the subgoals acquired in the preceding phase now become additional information items which must be retained.

The developmental mechanisms described in the chapter on BAIRN by Wallace, Klahr, and Bluff rely on analyzing and comparing multiple sequences of actions. The information required to do so is organized primarily at the level of "nodes", which are groups of

related productions. (Nodes may be loosely analogized to Anzai's subgoals, but are much more fundamental to the BAIRN architecture.) A "time line" represents the chronological sequence of node activations, along with the activating input, the resulting state, and a flag for cases of unexpected results. (Thus, information about expectations is an additional implicit aspect of the information demands of their learning model.) If a node is in "focal consciousness" (i.e., in the foreground of conscious attention), then information is also retained about the sequence of productions fired in the course of processing that node. In addition to this information, the learning mechanisms also make use of an "experience list" associated with each node, which contains the memory of action sequences that commonly follow the given node. BAIRN also makes use of information about the applicability conditions of nodes, and about their superordinate/subordinate relationships to other nodes.

The strategy transformation model described in Neches' chapter focuses on the refinement and optimization of procedures that are already well-defined. His approach postulates a set of heuristics for detecting opportunities to improve the cognitive efficiency of procedures. After analyzing the information requirements of these heuristics, he proposed eleven information categories utilized by his learning mechanisms.[15] Most of the cases described above also seem to fit into these categories:

1. *goals and procedures*;
2. *episodes* – the structure representing a single problem solution;
3. *events* – individual instances of goals with particular inputs;
4. *inputs* – internal concepts or representations of external states considered by a goal;
5. *results* – concepts or state representations resulting from a goal;
6. relationships between *goals* and their *subgoals*;
7. the *effort* associated with achieving a goal;
8. *temporal orderings* of events (or production firings);
9. *processing information* (size of working memory, presence in working memory, etc.);
10. *descriptive information* – assertions about properties associated with a concept;
11. *frequency information* – knowledge about how frequently a concept is accessed.

The work by Neches and others indicates the large variety of information that is prerequisite to learning. This also has implications for the sheer *amount* of information that a learning/performance system must be able to sift through. Models of performance, or even of particular learning mechanisms, can gloss over this issue by considering only the information relevant to the topic being modeled. However, a complete model of an information processing system must address the entire range of information categories.

[15] Implicitly, there is a twelfth category: information about production instantiations, i.e., linkages between concepts referenced in a production's condition side and to the concepts resulting from its action side.

model. We urge readers to consider the following chapters with respect to the checklist of issues that can be constructed from these two themes. The first theme was essentially oriented towards characterizing the methods necessary in a complete learning system. The second was concerned with the goals of learning systems.

(1) Under the rubric of "components of learning", we presented some assertions about *requirements* for learning systems. We argued that there were four sub-problems which all learning researchers must address – either by providing mechanisms in their model, or by narrowing their self-assigned mission so as to render the requirement moot. *Aggregation* is the problem of identifying the data from which learning will be done, separating signal from noise as it were. *Clustering* is the problem of developing extensional definitions for concepts based on those data (e.g., recognizing positive and negative instances of a concept). *Characterization* is the problem of developing intensional hypotheses or descriptions of concepts and classes identified in clustering, so that instances not previously encountered can be recognized and utilized. *Storage/Indexing* is the problem of saving those characterizations so that they can be retrieved and applied in appropriate situations.

(2) Under the various headings of "*correct vs. interesting*" or "*applicability vs. desirability*", we suggested that learning is focussed towards some purpose. Learning systems must address two concerns. They must ensure that the conditions of their new rules are (or eventually will become) correct, in the sense that the rules are applicable whenever appropriate and not applicable under any other circumstances. They must also ensure that the actions of these rules are useful, in the sense that there is some criteria by which the system is better off for having behaved according to its new rules rather than its old ones. Not everything that could be learned is worth learning; therefore, mechanisms that produce correct rules do not necessarily produce useful rules.

Introducing Learning Mechanisms into Production System Architectures

Operationalized into production system terms, a system has "learned" if, as a result of experience, it comes to apply a different production in some situation than it would have applied in the equivalent situation at an earlier point in its lifetime. Given the processing cycle of production system architectures, all learning mechanisms can be characterized as seeking to accomplish this goal by manipulating one or more of the following processes within that cycle: production matching, conflict resolution, and application.

We also presented several constraints to which learning systems are subject, and argued that the natures of both learning mechanisms and production system architectures are shaped by the need to interact in ways that satisfy these constraints. Among the constraints considered were *efficiency and adaptivity*, *even granularity* of rules, and *analyzability* of rules. In preceding sections, we indicated how these constraints have led modern production system theories in particular directions. These include favoring exhaustive pattern matching processes over other possible ways of stating and testing production conditions, moves towards refined conflict resolution techniques involving notions like "strength" of production rules, and an increased emphasis on building a sensitivity

to goals into the architecture for both conflict resolution and control of working memory contents.

We hope that the present chapter has provided a useful overview of production systems and their role in modeling learning and development. As we will see in the following chapters, different researchers have taken quite different paths in developing self-modifying production systems. However, underlying these differences are a common set of goals and a common framework, and it is these common concerns that we have attempted to present. We believe that the production system approach holds great promise for understanding the nature of human learning and development, and we hope that future researchers will build upon the excellent beginning described in the chapters that follow.

References

Anderson, J. R. (1976). *Language, memory, and thought.* Hillsdale, N.J.: Lawrence Erlbaum Associates.

Anderson, J. R. (1981). A theory of language acquisition based on general learning principles. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 165–170). Vancouver, B.C., Canada.

Anderson, J. R. (1983). *The Architecture of cognition.* Cambridge, Mass.: Harvard University Press.

Anderson, J. R., Greeno, J. G., Kline, P. J., & Neves, D. M. (1981). Acquisition of problem–solving skill. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, N. J.: Lawrence Erlbaum Associates.

Anderson, J. R., & Kline, P. J. (1979). A learning system and its psychological implications. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (pp. 16–21). Tokyo, Japan.

Anderson, J. R., Kline, P. J., & Beasley, C. M. (1978). *A theory of the acquisition of cognitive skills.* (Technical Report No. ONR 77–1) Department of Psychology, Yale University, New Haven, CT.

Anderson, J. R., Kline, P. J., & Lewis, C. (1977). A production system model for language processing. In P. Carpenter & M. Just (Eds.), *Cognitive processes in comprehension.* Hillsdale, N.J.: Lawrence Erlbaum Associates.

Anzai, Y. (1978). Learning strategies by computer. *Proceedings of the Second Biennial Conference of the Canadian Society for Computational Studies of Intelligence* (pp. 181–190). Toronto, Ontario, Canada.

Anzai, Y., & Simon, H. A. (1979). The theory of learning by doing. *Psychological Review, 86,* 124–140.

Baylor, G. W., Gascon, J., Lemoyne, G., & Pother, N. (1973). An information processing model of some seriation tasks. *Canadian Psychologist, 14,* 167–196.

Berwick, R. (1979). Learning structural descriptions of grammar rules from examples. *Proceedings of the Sixth International Conference on Artificial Intelligence* (pp. 56–58). Tokyo, Japan.

Brazdil, P. (1978). Experimental learning model. *Proceedings of the Third AISB/GI Conference* (pp. 46–50). Hamburg, West Germany.

Brown, J. S., & Burton, R. R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science, 2,* 155–192.

Brown, J. S., & VanLehn, K. (1980). Repair theory: a generative theory of bugs in procedural skill. *Cognitive Science, 4,* 379–427.

Brownston, L., Farrell, R., Kant, E., & Martin, N. (1985). *Programming expert systems in OPS5: An introduction to rule-based programming.* Reading, Mass.: Addison-Wesley.

Bundy, A. (1984). *What has learning got to do with expert systems?* (Research Paper No. 214) Department of Artificial Intelligence, University of Edinburgh.

Easterlin, J. D. & Langley, P. (1985). A framework for concept formation. *Proceedings of the Seventh Conference of the Cognitive Science Society* (pp. 267–271). Irvine, California.

Fisher, D. & Langley, P. (1985). Approaches to conceptual clustering. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 691–697). Los Angeles, California.

Forgy, C. L. (1979a). *OPS4 user's manual.* (Technical Report) Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Forgy, C. L. (1979b). *On the efficient implementation of production systems.* Dissertation, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Forgy, C. L. (1981). *OPS5 user's manual.* (Technical Report) Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Forgy, C. L. (1982). Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence, 19,* 17–37.

Forgy, C. L. (1984). *The OPS83 report.* (Technical Report) Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Forgy, C. L., & McDermott, J. (1976). *The OPS reference manual.* (Technical Report) Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Forgy, C. L., & McDermott, J. (1977). *The OPS2 reference manual.* (Technical Report) Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Hayes–Roth, F., & McDermott, J. (1978). An interference matching technique for inducing abstractions. *Communications of the ACM, 21,* 401–410.

Hedrick, C. (1976). Learning production systems from examples. *Artificial Intelligence, 7,* 21–49.

Hunter, I. M. L. (1968). Mental calculation. In P.C. Wason & P.N. Johnson–Laird (Eds.), *Thinking and reasoning.* Baltimore: Penguin Books.

Klahr, D. (1973). A production system for counting, subitizing, and adding. In W. G. Chase (Ed.), *Visual information processing.* New York: Academic Press.

Klahr, D., & Siegler, R. (1978). The representation of children's knowledge. In H. Reese & L. P. Lipsitt (Eds.), *Advances in child development,* Vol. 12. New York, Academic Press.

Klahr, D., & Wallace, J. G. (1972). Class inclusion processes. In S. Farnham–Diggory (Ed.), *Information processing in children.* New York, Academic Press.

Klahr, D., & Wallace, J. G. (1973). The role of quantification operators in the development of conservation of quantity. *Cognitive Psychology, 4,* 301–327.

Klahr, D., & Wallace, J. G. (1976). *Cognitive development: An information processing view.* Hillsdale, N. J.: Lawrence Erlbaum Associates.

Laird, J. E. (1983). *Universal subgoaling.* Dissertation, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Laird, J. E., Rosenbloom, P. S., & Newell, A. (1984). Towards chunking as a general learning mechanism. In *Proceedings of the National Conference on Artificial Intelligence* (pp. 188–192). Austin, Texas.

Langley, P. (1978). BACON.1: A general discovery system. *Proceedings of the Second Biennial Conference of the Canadian Society for Computational Studies of Intelligence* (pp. 173–180). Toronto, Ontario, Canada.

Langley, P. (1982). Language acquisition through error recovery. *Cognition and Brain Theory, 5*, 211–255.

Langley, P. (1983). Learning search strategies through discrimination. *International Journal of Man–Machine Studies, 18*, 513–541.

Langley, P., & Neches, R. T. (1981). *PRISM user's manual.* (Technical Report) Computer Science Department, Carnegie–Mellon University, Pittsburgh, PA.

Langley, P., Neches, R. T., Neves, D., & Anzai, Y. (1980). A domain–independent framework for learning procedures. *International Journal of Policy Analysis and Information Systems, 4*, 163–197.

Langley, P., & Ohlsson, S. (1984). Automated cognitive modeling. *Proceedings of the Fourth National Conference of the American Association for Artificial Intelligence* (pp. 193–197). Austin, Texas.

Langley, P., Ohlsson, S., Thibadeau, R., & Walter, R. (1984). Cognitive architectures and principles of behavior. *Proceedings of the Sixth Conference of the Cognitive Science Society* (pp. 244–247). Boulder, Colorado.

Langley, P. & Simon, H. A. (1981). The central role of learning in cognition. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, N.J.: Lawrence Erlbaum Associates.

Larkin, J. H. (1981). Enriching formal knowledge: A model for learning to solve textbook physics problems. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, N. J.: Lawrence Erlbaum Associates.

Lenat, D. B. (1977). Automated theory formation in mathematics. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (pp. 833–842). Cambridge, Mass.

Lewis, C. (1978). *Production system models of practice effects.* Dissertation, Department of Psychology, University of Michigan, Ann Arbor, Michigan.

Lewis, C. (1981). Skill in algebra. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, N. J.: Lawrence Erlbaum Associates.

McDermott, J. (1979). Learning to use analogies. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (pp. 568–576). Tokyo, Japan.

McDermott, J. (1982). R1: A rule–based configurer of computer systems. *Artificial Intelligence, 19*, 39–88.

McDermott, J., & Forgy, C. L. (1978). Production system conflict resolution strategies. In D. A. Waterman & F. Hayes–Roth (Eds.), *Pattern–directed inference systems.* Orlando, FL: Academic Press Inc.

Michalski, R. S. (1980). Pattern recognition as rule–guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 2*, 349–361.

Minsky, M. (1963). Steps toward artificial intelligence. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and thought.* New York: McGraw–Hill, Inc.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence, 18*, 203–226.

Mitchell, T. M., Utgoff, P., & Banerji, R. B. (1983). Learning problem solving heuristics by experimentation. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* Palo Alto, CA: Tioga Publishing Co.

Neches, R. (1981a). A computational formalism for heuristic procedure modification. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 283–288). Vancouver, B.C., Canada.

Neches, R. (1981b). *Models of heuristic procedure modification.* Dissertation, Department of Psychology, Carnegie–Mellon University, Pittsburgh, PA.

Neches, R. (1982). Simulation systems for cognitive psychology. *Behavior Research Methods and Instrumentation, 14*, 77–91.

Neves, D. M. (1978). A computer program that learns algebraic procedures by examining examples and working problems in a textbook. *Proceedings of the Second Biennial Conference of the Canadian Society for Computational Studies of Intelligence* (pp. 191–195). Toronto, Ontario, Canada.

Neves, D. M., & Anderson, J. R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, N. J.: Lawrence Erlbaum Associates.

Newell, A. (1967). *Studies in Problem Solving: Subject 3 on the Crypt–Arithmetic task Donald + Gerald = Robert.* (Technical Report) Center for the Study of Information Processing, Carnegie Institute of Technology, Pittsburgh, PA.

Newell, A. (1972). A theoretical exploration of mechanisms for coding the stimulus. In A. W. Melton & E. Martin (Eds.), *Coding processes in human memory.* Washington, Winston.

Newell, A. (1973). Production systems: Models of control structures. In W. G. Chase (Ed.), *Visual information processing.* New York: Academic Press.

Newell, A. (1980). Reasoning, problem solving, and decision processes: The problem space hypothesis. In R. Nickerson (Ed.), *Attention and performance. Volume VIII.* Hillsdale, N. J.: Lawrence Erlbaum Associates.

Newell, A., & McDermott, J. (1975). *PSG manual.* (Technical Report) Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Newell, A., & Rosenbloom, P. S. (1981). Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition.* Hillsdale, N. J.: Lawrence Erlbaum Associates.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, N.J.: Prentice–Hall, Inc.

Ohlsson, S. (1979). *PSS3 reference manual.* (Technical Report) Department of Psychology, University of Stockholm, Stockholm, Sweden.

Ohlsson, S. (1980a). *A possible path to expertise in the three–term series problem.* (Technical Report) Department of Psychology, University of Stockholm, Stockholm, Sweden.

Ohlsson, S. (1980b). *Competence and strategy in reasoning with common spatial concepts: A study of problem solving in a semantically rich domain.* Dissertation, Department of Psychology, University of Stockholm, Stockholm, Sweden.

Ohlsson, S. (1983). A constrained mechanism for procedural learning. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (pp. 426–428). Karlsruhe, West Germany.

Rosenbloom, P. S. (1979). *XAPS reference manual.* (Technical Report) Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Rosenbloom, P. S. (1983). *The chunking model of goal hierarchies: A model of practice and stimulus–response compatibility.* Dissertation, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Rychener, M. D. (1976). *Production systems as a programming language for artificial intelligence applications.* Dissertation, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Rychener, M. D. (1980). *OPS3 production system language: Tutorial and reference manual.* Unpublished manuscript, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Sauers, R., & Farrell, R. (1982). *GRAPES user's manual.* (Technical Report) Department of Psychology, Carnegie–Mellon University, Pittsburgh, PA.

Seibel, R. (1963). Discrimination reaction time for a 1,023 alternative task. *Journal of Experimental Psychology, 66,* 215–226.

Shortliffe, E. H. (1976). *MYCIN: Computer–based medical consultations.* New York: Elsevier.

Simon, D. P., & Simon, H. A. (1978). Individual differences in solving physics problems. In R. Siegler (Ed.), *Children's thinking: What develops?* Hillsdale, N. J.: Lawrence Erlbaum Associates.

Sleeman, D., Langley, P., & Mitchell, T. (1982). Learning from solution paths: An approach to the credit assignment problem. *AI Magazine, 3*, 48–52.

Thibadeau, R. (1982). CAPS: A language for modeling highly–skilled knowledge–intensive behavior. *Proceedings of the National Conference on the Use of On–line Computers in Psychology.* Minneapolis, Minnesota.

Thibadeau, R., Just, M. A., & Carpenter, P. A. (1982). A model of the time course and content of reading. *Cognitive Science, 6*, 157–203.

Waterman, D. A. (1970). Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence, 1*, 121–170.

Waterman, D. A. (1975). Adaptive production systems. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (pp. 296–303). Tbilisi, USSR.

Waterman, D. A., & Newell, A. (1972). *Preliminary results with a system for automatic protocol analysis.* (Technical Report CIP No. 211) Department of Psychology, Carnegie–Mellon University, Pittsburgh, PA.

Winston, P. H. (1975). Learning structural descriptions from examples. In P. H. Winston (Ed.), *The psychology of computer vision.* New York: McGraw–Hill.

Young, R. M. (1976). *Seriation by children: An artificial intelligence analysis of a Piagetian task.* Basel: Birkhauser.

Young, R. M., & O'Shea, T. (1981). Errors in children's subtraction. *Cognitive Science, 5*, 153–177.